03/09/2023
Sunday

* BITWISE OPERATORS -

If operations are to be done at the binary level or bit-level, here bitwise operators comes into use.
These operations are necessary because the ALU (Arithmetical logical unit) present in the CPU carries out arithmetic operations at the bit-level.

Bitwise operators can operate only on int and char data types and not on float or double data types.

TYPES OF BITWISE OPERATORS -

(i) Bitwise AND (&)
(ii) Bitwise OR (|)
(iii) Bitwise NOT (~)
(iv) Bitwise XOR (^)

• Truth Table for Bitwise AND -

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

__/__/____

- **Truth Table for Bitwise OR –**

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

- **Truth Table for Bitwise NOT or Complement –**

A	Output
0	1
1	0

- **Truth Table for Bitwise XOR –**

A	B	$\overline{A}$	$\overline{B}$	$\overline{A}B$	$A\overline{B}$	$A\overline{B} + \overline{A}B$ OR $A \oplus B$
0	0	1	1	0	0	0
0	1	1	0	1	0	1
1	0	0	1	0	1	1
1	1	0	0	0	0	0

A	B	Output ($A \oplus B$)
0	0	0
0	1	1
1	0	1
1	1	0

* Bitwise AND-

Eg-1 int main(){ → Why?

```
cout<<(2&3);
}
```

Output = 2

2=10 ; 3=11

```
 1 0
 1 1
-----
 1 0
```

AND(&)= 10 which is 2.

Eg-2 int main(){ → Why?

```
cout<<(5&10);
}
```

Output = 0

5=101 ; 10=1010

```
  0 1 0 1
  1 0 1 0
----------
  0 0 0 0
```

AND(&) = 0000 which is 0.

* Bitwise OR -

Eg-1 int main(){ → Why?

```
cout<<(5|10);
}
```

Output = 15

5=101 ; 10=1010

```
  0 1 0 1
  1 0 1 0
----------
  1 1 1 1
```

OR(|) = 1111 which is 15.

Eg-2 int main(){ → Why?

```
cout<<(2|3);
}
```

Output = 3

2=10 ; 3=11

```
 1 0
 1 1
-----
 1 1
```

OR(|) 11 which is 3.

__/__/____

* **Bitwise NOT -**

Eg-
```
int main ( ) {
cout << (~1);
}
```

Output = -2

Why ?

1 = 00000000 00000000 00000000 00000001

(~1) = 11111111 11111111 11111111 11111110

↓

This MSB shows that it is a -ve number.

2's complement of (~1) -

```
  00000000  00000000  00000000  00000001
                                    +1
  --------  --------  --------  --------
  00000000  00000000  00000000  00000010
```

which is binary representation of 2.

That's why cout << (~1); gives output as -2.

* Bitwise XOR –

Eg-1

```
int main(){
cout<<(5^5);
}
```

Output = 0

Why?

5 = 101

```
  101
  101
  ---
  000
```
XOR (^)

which is 0.

Eg-2

```
int main(){
cout<<(5^10);
}
```

Output = 15

Why?

5 = 101 ; 10 = 1010

```
 0101
 1010
 ----
 1111
```
XOR (^)

which is 15.

Eg-3

```
int main(){
cout<<(4^7);
}
```

Output = 3

Why?

4 = 100 ; 7 = 111

```
  100
  111
  ---
  011
```
XOR (^)

which is 3

__/__/____

Eg→4 int main( ){

cout<< (5^-5);

}

Output = -2

Why?

5 = 00000000 00000000 00000000 00000101

-5 = 11111111 11111111 11111111 11111011

XOR(^) (1)1111111 11111111 11111111 11111110

↓

This MSB shows that XOR of 5 and -5 is a -ve number.

2's complement of (5^-5) →

00000000 00000000 00000000 00000001

+1

00000000 00000000 00000000 00000010

which is binary representation of 2

That's why output of cout<<(5^-5); is -2.

__/__/____

* LEFT AND RIGHT SHIFT OPERATOR –

- Left shift operator → "<<"

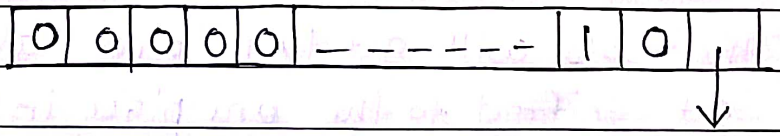
```
int main( ){
    int a= 2;                          output = 4
    cout << (a<<1)<<endl;
}
```

What happens inside memory?

(a= 2) = | 0 | 0 | 0 | 0 | 0 | ------ | 1 | 0 |

a<<1 i.e. a is left shifted by 1 bit.

Here, the MSB will get lost and all other bits gets shifted to the one place in the left.



This empty place will be filled by 0.

| 0 | 0 | 0 | 0 | 0 | ------ | 1 | 0 | 0 |

This is binary representation of 4 which we got after left shifting of (a) by 1 bit.

__/__/____

NOTE - When we left shift a variable (a) n times, the output we will get is equal to →

$a * 2^n$ → This is the generalised statement for unsigned integer(+ve) values only.

- **Right Shift operator → ">>"**

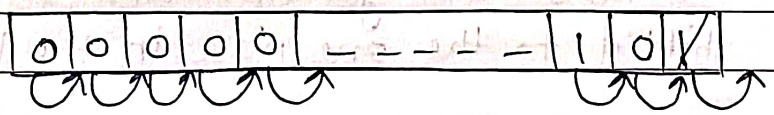
```
int main(){

  int n= 5;                    output= 2
  cout<<(n>>1)<<endl;
}
```

What happens inside memory?

(n=5) = 

n>>1 i.e. n is right shifted by 1 bit



Here, the LSB will get lost and all other bits will get shifted to the one place in right.

This empty place will be filled by 0.

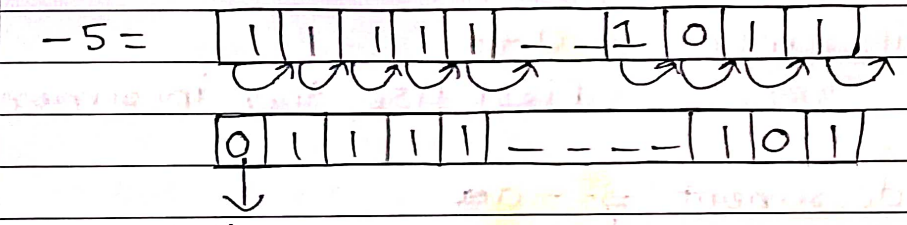
This is the binary representation of 2 which we got after right shifting of (n) by 1 bit.

__/__/____

NOTE – When we right shift a variable (a) n times, the output we get is equal to →

$a * 2^{-n}$ → This is the generalised statement for unsigned (+ve) integer values only.

* Let we have to do right shifting in case of a negative number →



The MSB shows that it is a +ve number.

BUT,

In Case of unsigned integer, right shifting is giving positive large number

AND

In Case of signed integer, compiler is handling the right shifting by giving -ve value for the -ve number.

__/__/____

NOTE- When we left or right shift a value by a -ve number, then it throws warning and gives a garbage value.
Remember, it will not show error.

* PRE / POST INCREMENT / DECREMENT -

- Pre-increment ⇒ ++a
  First increment, then use.
- Post-increment ⇒ a++
  First use, then increment.
- Pre-decrement ⇒ --a
  First decrement, then use.
- Post-decrement ⇒ a--
  First use, then decrement.

__/__/____

For instance,

```
int main() {

int a= 10;

// Pre increment
 cout << (++a); --> 11
 cout << (++a) * 10; --> 120

// Post increment
  cout<< (a++); --> 12
  cout<< (a++) * 10; --> 130

// Pre decrement
  cout<< (--a); --> 13
  cout << (--a) * 10; --> 120

// Post decrement
  cout << (a--); --> 12
  cout<< (a--) * 10; --> 110

  cout << (a); --> 10

}
```

* BREAK AND CONTINUE STATEMENTS –

- **Break –** The Break statement is used to terminate the loop. If we want to terminate the loop based on some conditions, then we use Break statement there.

Syntax → Break;

Example –

```
int main( )
{
  for (int i=0 ; i<5 ; i++)
    {
      if (i==4){
        Break;
      }
    cout<< i<< endl;
    }
}
```

Output –

```
0
1
2
3
```

- **Continue** – The continue statement is used to skip the current iteration of the loop. If we want to skip any iteration based on some conditions, then we use continue statement these.

Syntax – continue;

Example –

```
int main( )
{
  for (int i=0 ; i<=5 ; i++)
    {
      if (i==4) {
        continue;
      }
    cout<< i << endl;
    }
}
```

Output –

```
0
1
2
3
5
```

* **VARIABLE SCOPING** - On basis of variable scoping, variables have two types →

  - Local Variables
  - Global variables

Local variables - local variables are accessible and updated in its scope only. Outside that scope, neither we can access nor update that variable.

for instance,

```
for(int i=0 ; i<n ; i++)
  {
    cout << i <<endl;
  }
cout << i <<endl; → This will show error.
```

The scope of the variable i is only inside the curly braces and outside it, we can't access i.

Global variables - Global variables are accessible anywhere in the program. These variables are initialised or declared outside the main function generally. But Global variables are considered as bad practice.

__/__/____

For instance,

```
#include<iostream>
using namespace std;

int a=5;

int main()
{
}
```

Here, the variable a is a global variable and can be accessed anywhere.

NOTE - (1) We can update a variable but can't re-declare it.

(2) We can re-declare a variable inside the nested statements, if-else or loops.

For instance,

```
if (true){
  int a=5;
  if (true){
    int a=10;
  }
}
```